


[Index](#) | [Rooms](#) | [Flags](#) | [Battles](#) | [Save structure](#) | [Monsters](#) | [Items](#) | [Papyrus & Undyne calls](#) | [Fun events](#) | [Debug mode](#) | [Unpacking \(original\)](#) | [Unpacking \(w/ corrections\)](#) | [Decompilation \(original\)](#) | [Decompilation \(w/ corrections\)](#) | [Sound effects](#) | [Sprites](#) | [Strings](#) | [Endings map](#) | 

Data types

Type	Length	Description
Double	8	
Int64	8	
Float	4	
Int32	4	
Bool	4	True if 1, else false
Int24	3	
Int16	2	
TypePair	1	Each type is 4-bit long
String	4	Index of the string in the Strg chunk.

Enums

DataType : Byte

- *Double*
- *Float*
- *Int32*
- *Int64*
- *Boolean*
- *Variable*
- *String*
- *Instance*
- *Int16 = 0x0f*

InstanceType : Int16

- *StackTopOrGlobal = 0*
- *Self = -1*
- *Other = -2*
- *All = -3*
- *Noone = -4*
- *Global = -5*

```

└─Unknown = -6 // TODO
└─Local = -7 // script-scope local var
└─ObjectSpecific // anything > 0 //If it's none of the given values, it represents a GameObjectIndex
VariableType : Byte
└─Array
└─StackTop = 0x80
└─Normal = 0xA0
└─Unknown = 0xE0 // room-scope vars? (only found in RoomCC stuff)
ComparisonType
└─LowerThan = 1
└─LTOrEqual = 2
└─Equal = 3
└─Inequal = 4
└─GTOrEqual = 5
└─GreaterThan = 6

```

Architecture

The interpreter is stack-based. Data is pushed and popped from the stack.

Variables and arrays

When parsing a Variable, if the Type is VariableType.Array: the index is at the stack top and has to be popped, and is followed by the Int16 -5, which also has to be popped. When an array is pushed and the Dup instruction occurs, the index is also duplicated. When Pop's or Push's InstanceType is InstanceType.StackTopOrGlobal, if Type is VariableType.StackTop then one additional value, representing the instance, will be popped from the stack.

References

Each ReferenceDefinition has a pointer to memory address of the first instruction that accesses it. In the second block of the instruction, there is the offset (calculated in blocks, not bytes) to the next occurrence of the reference. Note that the Type of the reference can change between different instances of said reference; for instance in one case an array item can be accessed, while in case the same array is accessed as a whole. Functions don't have a Type, only the offset to the next occurrence.

Instructions

Each instruction is composed of one or more 32-bit blocks. The most significant byte is the opcode.

```

Reference
└─Type : VariableType
└─NextOccurrenceOffset : Int24

```

```

Variable : Reference
DoubleTypeInstruction
├─ padding : Int16
├─ Types : TypePair
└─ OpCode : Byte
ComparisonInstruction
├─ padding : Byte
├─ Comparison : ComparisonType
├─ Types : TypePair
└─ OpCode : Byte
SingleTypeInstruction
├─ padding : Int16
├─ Type : DataType
└─ OpCode : Byte
GotoInstruction
├─ Offset : Int24
└─ OpCode : Byte

```

For bytecode version 0xE, the opcode table is as follows:

```

0x03: Conv : DoubleTypeInstruction //Push((Types.Second)Pop)
0x04: Mul : DoubleTypeInstruction //Push(Pop() * Pop())
0x05: Div : DoubleTypeInstruction //Push(Pop() / Pop())
0x06: Rem : DoubleTypeInstruction //Push(Pop() % Pop())
0x06: Rem : DoubleTypeInstruction //Push(Remainder(Pop(), Pop()))
0x07: Rem : DoubleTypeInstruction //Push(Pop() % Pop())
0x08: Add : DoubleTypeInstruction //Push(Pop() + Pop())
0x09: Sub : DoubleTypeInstruction //Push(Pop() - Pop())
0x0a: And : DoubleTypeInstruction //Push(Pop() & Pop())
0x0b: Or : DoubleTypeInstruction //Push(Pop() | Pop())
0x0e: Not : DoubleTypeInstruction //Push(!Pop())
0x11: Slt : DoubleTypeInstruction //Push(Pop() < Pop())
0x12: Sle : DoubleTypeInstruction //Push(Pop() <= Pop())
0x13: Seq : DoubleTypeInstruction //Push(Pop() == Pop())
0x14: Sne : DoubleTypeInstruction //Push(Pop() != Pop())
0x15: Sge : DoubleTypeInstruction //Push(Pop() >= Pop())
0x16: Sgt : DoubleTypeInstruction //Push(Pop() > Pop())
0x41: Pop //Instance.Destination = Pop(); //ATTENTION: if Types.First is Int32, the value will be on top of the stack, even
before array access parameters!
├─ Block1
│   ├── Types : TypePair
│   ├── Instance : InstanceType
│   └─ OpCode : OpCode
└─ Block2
    └─ Destination : Variable
0x82: Dup : SingleTypeInstruction //Push(Peek())
0x9d: Ret : SingleTypeInstruction //return Pop()
0x9e: Exit : SingleTypeInstruction //return;
0x9f: Popz : SingleTypeInstruction //Pop();

```

```

0xb7: B : GotoInstruction //goto Index + Offset*4;
0xb8: Bt : GotoInstruction //if (Pop()) goto Index + Offset*4;
0xb9: Bf : GotoInstruction //if (!Pop()) goto Index + Offset*4;
0xbb: Pushenv : GotoInstruction
0xbc: Popenv : GotoInstruction
0xc0: Push //Push(Value)
├─ Value : Variant //Depends on Type. Types longer than Int16 take one or more extra blocks. If Type is Variable, the
│   Instance type is an Int16 in Block1 and the Reference is in Block2
├─ Type : DataType
└─ OpCode : OpCode
0xda: Call //Function(arg0, arg1, ..., argn) where arg = Pop() and n = ArgumentsCount
├─ Block1
│   ├── ArgumentsCount : Int16
│   ├── ReturnType : DataType
│   └─ OpCode : OpCode
├─ Block2
│   └─ Function : Reference
0xff: Break //Invalid access guard?
├─ Signal : Int16
├─ Type : DataType
└─ OpCode : OpCode

```

The following is valid for version 0xF:

```

0x07: Conv : DoubleTypeInstruction //Push((Types.Second)Pop)
0x08: Mul : DoubleTypeInstruction //Push(Pop() * Pop())
0x09: Div : DoubleTypeInstruction //Push(Pop() / Pop())
0x0A: Rem : DoubleTypeInstruction //Push(Remainder(Pop(), Pop()))
0x0B: Mod : DoubleTypeInstruction //Push(Pop() % Pop())
0x0C: Add : DoubleTypeInstruction //Push(Pop() + Pop())
0x0D: Sub : DoubleTypeInstruction //Push(Pop() - Pop())
0x0E: And : DoubleTypeInstruction //Push(Pop() & Pop())
0x0F: Or : DoubleTypeInstruction //Push(Pop() | Pop())
0x10: Xor : DoubleTypeInstruction //Push(Pop() ^ Pop())
0x11: Neg : SingleTypeInstruction //Push(-Pop()) // two's complement
0x12: Not : SingleTypeInstruction //Push(~Pop()) // one's complement
0x13: Shl : DoubleTypeInstruction //Push(Pop() << Pop())
0x14: Shr : DoubleTypeInstruction //Push(Pop() >>= Pop())
0x15: Cmp : ComparisonInstruction //Push(Pop() `cmp` Pop()) // The actual comparison depends on the ComparisonType
0x45: Pop //Instance.Destination = Pop(); //ATTENTION: if Types.First is Int32, the value will be on top of the stack, even
before array access parameters!
├─ Block1
│   ├── Types : TypePair
│   ├── Instance : InstanceType
│   └─ OpCode : OpCode
├─ Block2
│   └─ Destination : Variable
// NOTE: a "magic array pop" instruction (06 00 5F 45) indicates something unknown as of yet.

```

```

0x86: Dup : SingleTypeInstruction //Push(Peek())
0x9C: Ret : SingleTypeInstruction //return Pop()
0x9D: Exit : SingleTypeInstruction //return;
0x9E: Popz : SingleTypeInstruction //Pop();
0xB6: B : GotoInstruction //goto Index + Offset*4;
0xB7: Bt : GotoInstruction //if (Pop()) goto Index + Offset*4;
0xB8: Bf : GotoInstruction //if (!Pop()) goto Index + Offset*4;
0xBA: PushEnv : GotoInstruction
0xBB: PopEnv : GotoInstruction
0xC0: PushCst //Push(Value) // push constant
0xC1: PushLoc //Push(Value) // push local
0xC2: PushGlb //Push(Value) // push global
0xC3: PushVar //Push(Value) // push other variable
0x84: PushI16 //Push(Value) // push int16
// NOTE: the push type is preserved from the previous version, so the program can check for any of these values instead of
// every kind separate and then checking the actual type
├─ Value : Variant //Depends on Type. Types longer than Int16 take one or more extra blocks. If Type is Variable, the
│   Instance type is an Int16 in Block1 and the Reference is in Block2
├─ Type : DataType
└─ OpCode : OpCode
0xD9: Call //Function(arg0, arg1, ..., argn) where arg = Pop() and n = ArgumentsCount
├─ Block1
│   ├── ArgumentsCount : Int16
│   ├── ReturnType : DataType
│   └─ OpCode : OpCode
├─ Block2
│   └─ Function : Reference
└─
0xFF: Break //Invalid access guard?
├─ Signal : Int16
├─ Type : DataType
└─ OpCode : OpCode

```

After parsing

If you want to decompile the code in high level language, you'll have to do some structuring. Some graph theory is required to get good output. There are while, do..while, for loops and special Repeat statements (essentially For loops without declared variables); no improper loops, but breaks and continues. There are ifs, if..else's and switch..case's. Also considering that there are returns and exits, you'll have to heuristically simulate the stack and branch it when the control flow breaks. For any questions or if I've made mistakes, message me on [Reddit](#).